6 Mapping URLs to endpoints using routing

This chapter covers

- Mapping URLs to endpoint handlers
- Using constraints and default values to match URLs
- Generating URLs from route parameters

In chapter 5 you learned how to define minimal APIs, how to return responses, and how to work with filters and route groups. One crucial aspect of minimal APIs that we touched on only lightly is how ASP.NET Core selects a specific endpoint from all the handlers defined, based on the incoming request URL. This process, called *routing*, is the focus of this chapter.

This chapter begins by identifying the need for routing and why it's useful. You'll learn about the endpoint routing system introduced in ASP.NET Core 3.0 and why it was introduced, and explore the flexibility routing can bring to the URLs you expose.

The bulk of this chapter focuses on the route template syntax and how it can be used with minimal APIs. You'll learn about features such as optional parameters, default parameters, and constraints, as well as how to extract values from the URL

automatically. Although we're focusing on minimal APIs in this chapter, the same routing system is used with Razor Pages and Model-View-Controller (MVC), as you'll see in chapter 14.

In section 6.4 I describe how to use the routing system to *generate* URLs, which you can use to create links and redirect requests for your application. One benefit of using a routing system is that it decouples your handlers from the underlying URLs they're associated with. You can use URL generation to avoid littering your code with hard-coded URLs like `/product/view/3`. Instead, you can generate the URLs at runtime, based on the routing system. This approach makes changing the URL for a given endpoint easier: instead of your having to hunt down every place where you used the endpoint's URL, the URLs are updated for you automatically, with no other changes required.

By the end of this chapter, you should have a much clearer understanding of how an ASP.NET Core application works. You can think of routing as being the glue that ties the middleware pipeline to endpoints. With middleware, endpoints, and routing under your belt, you'll be writing web apps in no time!

6.1 What is routing?

Routing is the process of mapping an incoming request to a method that will handle it. You can use routing to control the URLs you expose in your application. You can also use routing to enable powerful features such as mapping multiple URLs to the same handler and automatically extracting data from a request's URL.

In chapter 4 you saw that an ASP.NET Core application contains a middleware pipeline, which defines the behavior of your application. Middleware is well suited to handling both cross-cutting concerns, such as logging and error handling, and narrowly focused requests, such as requests for images and CSS files.

To handle more complex application logic, you'll typically use the `EndpointMiddleware` at the end of your middleware pipeline. This middleware can handle an appropriate request by invoking a method known as a handler and using the result to generate a response. Previous chapters described using minimal API endpoint handlers, but there are other types of handlers, such as MVC action methods and Razor Pages, as you'll learn in part 2 of this book.

One aspect that I've glossed over so far is *how* the `EndpointMiddleware` selects which handler executes when you receive a request. What makes a request appropriate for a given handler? The process of mapping a request to a handler is *routing*.

> **DEFINITION** *Routing* in ASP.NET Core is the process of selecting a specific handler for an incoming HTTP request. In minimal APIs, the handler is the endpoint handler associated with a route. In Razor Pages, the handler is a page handler method defined in a Razor Page. In MVC, the handler is an action method in a controller.

In chapters 3 to 5, you saw several simple applications built with minimal APIs. In chapter 5, you learned the basics of routing for minimal APIs, but it's worth exploring

why routing is useful as well as how to use it. Even a simple URL path such as `/person` uses routing to determine which handler should be executed, as shown in figure 6.1.

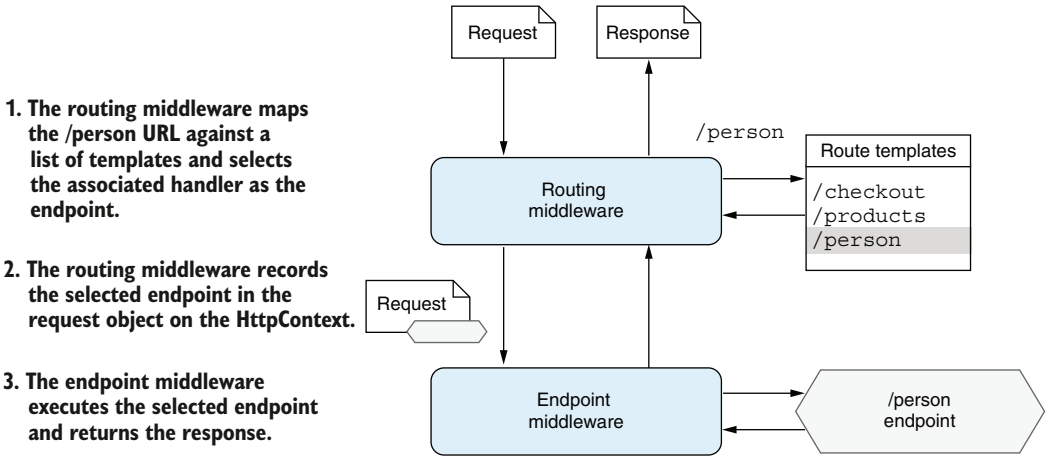

Figure 6.1 The router compares the request URL with a list of configured route templates to determine which handler to execute.

On the face of it, that seems pretty simple. You may wonder why I need a whole chapter to explain that obvious mapping. The simplicity of the mapping in this case belies how powerful routing can be. If this approach, using a direct comparison with static strings, were the only one available, you'd be severely limited in the applications you could feasibly build.

Consider an e-commerce application that sells multiple products. Each product needs to have its own URL, so if you were using a purely static routing system, you'd have only two options:

- *Use a different handler for every product in your product range.* That approach would be unfeasible for almost any realistically sized product range.
- *Use a single handler, and use the query string to differentiate among products.* This approach is much more practical, but you'd end up with somewhat-ugly URLs, such as `"/product?name=big-widget"` or `"/product?id=12"`.

DEFINITION The *query string* is part of a URL containing additional data that doesn't fit in the path. It isn't used by the routing infrastructure to identify which handler to execute, but ASP.NET Core can extract values from the query string automatically in a process called *model binding*, as you'll see in chapter 7. The query string in the preceding example is `id=12`.

With routing, you can have a *single* endpoint handler that can handle *multiple* URLs without having to resort to ugly query strings. From the point of the view of the endpoint handler, the query string and routing approaches are similar; the handler

returns the results for the correct product dynamically as appropriate. The difference is that with routing, you can completely customize the URLs, as shown in figure 6.2. This feature gives you much more flexibility and can be important in real-life applications for search engine optimization (SEO).

NOTE With the flexibility of routing, you can encode the hierarchy of your site properly in your URLs, as described in Google's SEO starter guide at http://mng.bz/EQ2J.

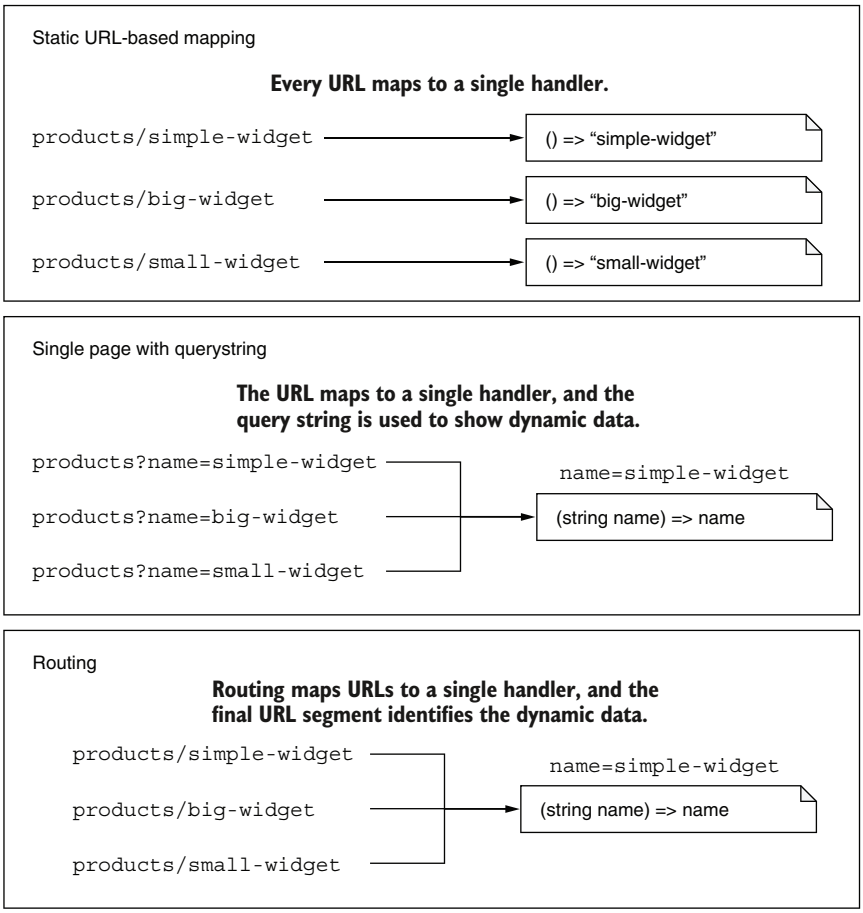

Figure 6.2 If you use static URL-based mapping, you need a different handler for every product in your product range. With a query string, you can use a single handler, and the query string contains the data. With routing, multiple URLs map to a single handler, and a dynamic parameter captures the difference in the URL.

As well as enabling dynamic URLs, routing fundamentally decouples the URLs in your application from the definition of your handlers.

File-system based routing

In one alternative to routing, the location of a handler on disk dictates the URL you use to invoke it. The downside of this approach is that if you want to change an exposed URL, you also need to change the location of the handler on disk.

This file-based approach may sound like a strange choice, but it has many advantages for some apps, primarily in terms of simplicity. As you'll see in part 2, Razor Pages is partially file-based but also uses routing to get the best of both worlds!

With routing it's easy to modify your exposed URLs without changing any filenames or locations. You can also use routing to create friendlier URLs for users, which can improve discovery and "hackability." All of the following routes could point to the same handler:

- `/rates/view/1`
- `/rates/view/USD`
- `/rates/current-exchange-rate/USD`
- `/current-exchange-rate-for-USD`

This level of customization isn't often necessary, but it's quite useful to have the capability to customize your app's URLs when you need it. In the next section we'll look at how routing works in practice in ASP.NET Core.

6.2 Endpoint routing in ASP.NET Core

In this section I describe how endpoint routing works in ASP.NET Core, specifically with respect to minimal APIs and the middleware pipeline. In chapter 14 you'll learn how routing is used with Razor Pages and the ASP.NET Core MVC framework.

Routing has been part of ASP.NET Core since its inception, but it has been through some big changes. In ASP.NET Core 2.0 and 2.1, routing was restricted to Razor Pages and the ASP.NET Core MVC framework. There was no dedicated routing middleware in the middleware pipeline; routing happened only within Razor Pages or MVC components.

Unfortunately, restricting routing to the MVC and Razor Pages infrastructure made some things a bit messy. Some cross-cutting concerns, such as authorization, were restricted to the MVC infrastructure and were hard to use from other middleware in your application. That restriction caused inevitable duplication, which wasn't ideal.

ASP.NET Core 3.0 introduced a new routing system: *endpoint routing*. Endpoint routing makes the routing system a more fundamental feature of ASP.NET Core and no longer ties it to the MVC infrastructure. Now Razor Pages, MVC, and other middleware can all use the same routing system. .NET 7 continues to use the same endpoint routing system, which is integral to the minimal API functionality that was introduced in .NET 6.

Endpoint routing is fundamental to all but the simplest ASP.NET Core apps. It's implemented with two pieces of middleware, which you've already seen:

- `EndpointRoutingMiddleware`—This middleware chooses which registered endpoints execute for a given request at runtime. To make it easier to distinguish between the two types of middleware, I'll be referring to this middleware as the `RoutingMiddleware` throughout this book.
- `EndpointMiddleware`—This middleware is typically placed at the end of your middleware pipeline. The middleware *executes* the endpoint selected by the `RoutingMiddleware` at runtime.

You register the endpoints in your application by calling `Map*` functions on an `IEndpointRouteBuilder` instance. In .NET 7 apps, this instance typically is a `WebApplication` instance but doesn't have to be, as you'll see in chapter 30.

> **DEFINITION** An *endpoint* in ASP.NET Core is a handler that returns a response. Each endpoint is associated with a URL pattern. Depending on the type of application you're building, minimal API handlers, Razor Page handlers, or MVC controller action methods typically make up the bulk of the endpoints in an application. You can also use simple middleware as an endpoint or you could use a health-check endpoint, for example.

`WebApplication` implements `IEndpointRouteBuilder`, so you can register endpoints on it directly. Listing 6.1 shows how you'd register several endpoints:

- A minimal API handler using `MapGet()`, as you've seen in previous chapters.
- A health-check endpoint using `MapHealthChecks()`. You can read more about health checks at http://mng.bz/N2YD.
- All Razor Pages endpoints in the application using `MapRazorPages()`. You'll learn more about routing with Razor Pages in chapter 14.

Listing 6.1 Registering multiple endpoints with `WebApplication`

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddHealthChecks();
builder.Services.AddRazorPages();

WebApplication app = builder.Build();

app.MapGet("/test", () => "Hello world!");
app.MapHealthChecks("/healthz");
app.MapRazorPages();

app.Run();
```

Adds the services required by the health-check middleware and Razor Pages

Registers a minimal API endpoint that returns "Hello World!" at the route /test

Registers a health-check endpoint at the route /healthz

Registers all the Razor Pages in your application as endpoints

Each endpoint is associated with a *route template* that defines which URLs the endpoint should match. You can see two route templates, `"/healthz"` and `"/test"`, in listing 6.1.

> **DEFINITION** A *route template* is a URL pattern that is used to match against request URLs, which are strings of fixed values, such as `"/test"` in the previous listing. They can also contain placeholders for variables, as you'll see in section 6.3.

The `WebApplication` stores the registered routes and endpoints in a dictionary that's shared by the `RoutingMiddleware` and the `EndpointMiddleware`.

> **TIP** By default, `WebApplication` automatically adds the `RoutingMiddleware` to the *start* of the middleware and `EndpointMiddleware` to the *end* of the middleware pipeline, though you can override the location in the pipeline by calling `UseRouting()` or `UseEndpoints()`. See section 4.2.3 for more details about automatically added middleware.

At runtime, the `RoutingMiddleware` compares an incoming request with the routes registered in the dictionary. If the `RoutingMiddleware` finds a matching endpoint, it makes a note of which endpoint was selected and attaches that to the request's `HttpContext` object. Then it calls the next middleware in the pipeline. When the request reaches the `EndpointMiddleware`, the middleware checks to see which endpoint was selected and executes the endpoint (and any associated endpoint filters), as shown in figure 6.3.

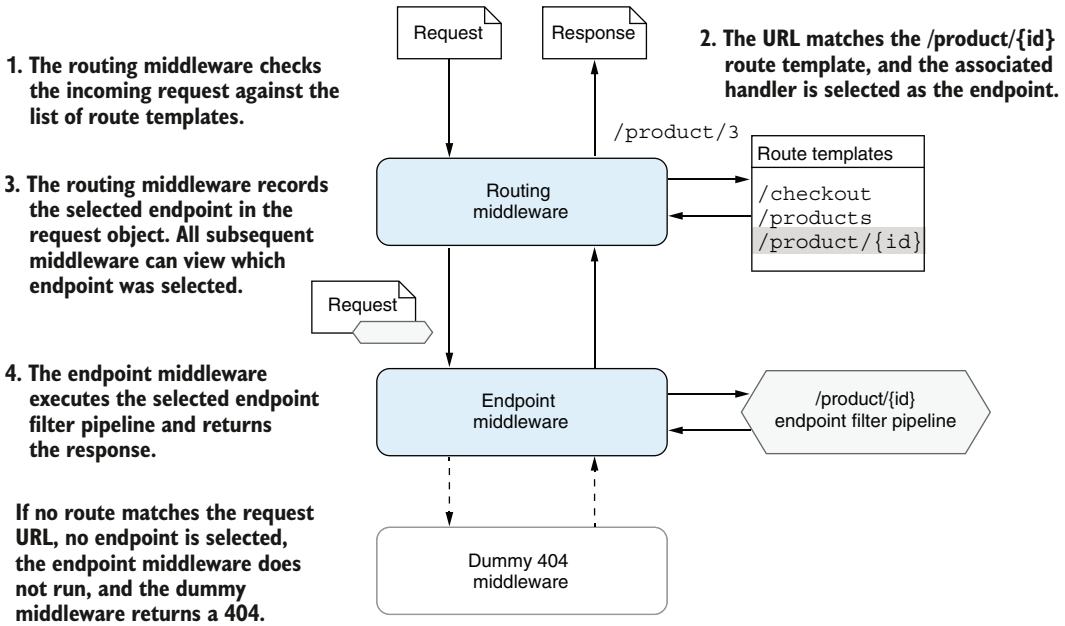

Figure 6.3 Endpoint routing uses a two-step process. The `RoutingMiddleware` selects which endpoint to execute, and the `EndpointMiddleware` executes it. If the request URL doesn't match a route template, the endpoint middleware won't generate a response.

If the request URL *doesn't* match a route template, the `RoutingMiddleware` doesn't select an endpoint, but the request still continues down the middleware pipeline. As no endpoint is selected, the `EndpointMiddleware` silently ignores the request and passes it to the next middleware in the pipeline. The `EndpointMiddleware` is typically the final middleware in the pipeline, so the "next" middleware is normally the dummy middleware that always returns a `404 Not Found` response, as you saw in chapter 4.

> **TIP** If the request URL doesn't match a route template, no endpoint is selected or executed. The whole middleware pipeline is still executed, but typically a `404` response is returned when the request reaches the dummy `404` middleware.

The advantage of having two separate pieces of middleware to handle this process may not be obvious at first blush. Figure 6.3 hinted at the main benefit: all middleware placed after the `RoutingMiddleware` can see which endpoint is *going* to be executed before it is.

> **NOTE** Only middleware placed after the `RoutingMiddleware` can detect which endpoint is going to be executed.

Figure 6.4 shows a more realistic middleware pipeline in which middleware is placed both *before* the `RoutingMiddleware` and *between* the `RoutingMiddleware` and the `EndpointMiddleware`.

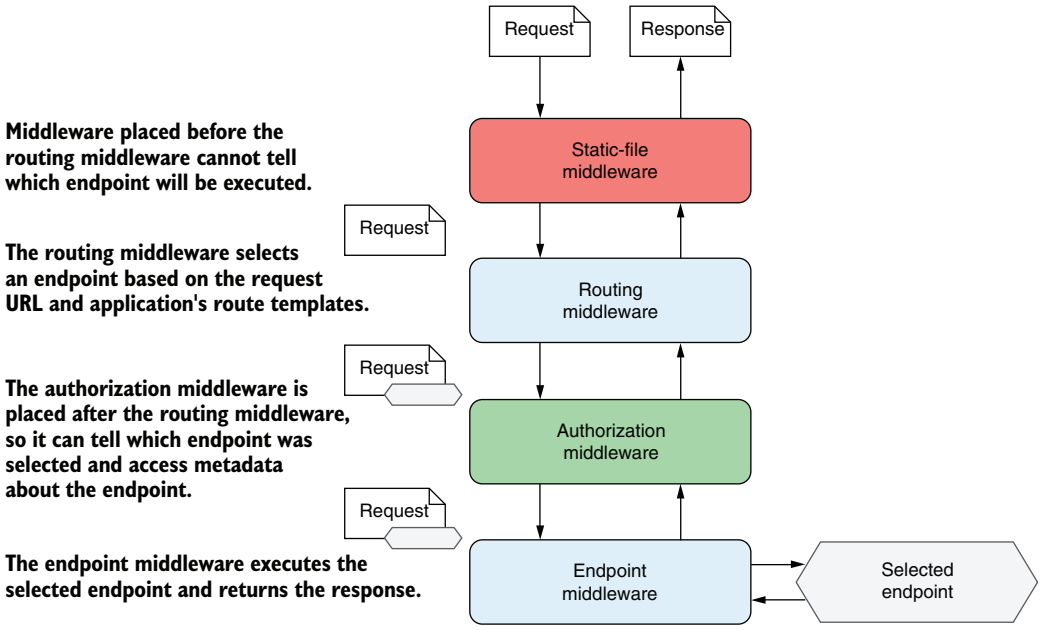

Figure 6.4 Middleware placed before the routing middleware doesn't know which endpoint the routing middleware will select. Middleware placed between the routing middleware and the endpoint middleware can see the selected endpoint.

The `StaticFileMiddleware` in figure 6.4 is placed before the `RoutingMiddleware`, so it executes before an endpoint is selected. Conversely, the `AuthorizationMiddleware` is placed after the `RoutingMiddleware`, so it can tell which minimal API endpoint will be executed eventually. In addition, it can access certain metadata about the endpoint, such as its name and the permissions required to access it.

> **TIP** The `AuthorizationMiddleware` needs to know which endpoint will be executed, so it must be placed after the `RoutingMiddleware` and before the `EndpointMiddleware` in your middleware pipeline. I discuss authorization in more detail in chapter 24.

It's important to remember the different roles of the two types of routing middleware when building your application. If you have a piece of middleware that needs to know which endpoint (if any) a given request will execute, you need to make sure to place it after the `RoutingMiddleware` and before the `EndpointMiddleware`.

> **TIP** If you want to place middleware before the `RoutingMiddleware`, such as the `StaticFileMiddleware` in figure 6.4, you need to override the automatic middleware added by `WebApplication` by calling `UseRouting()` at the appropriate point in your middleware pipeline. See listing 4.3 in chapter 4 for an example.

I've covered how the `RoutingMiddleware` and `EndpointMiddleware` interact to provide routing capabilities in ASP.NET Core, but we've looked at only simple route templates so far. In the next section we'll look at some of the many features available with route templates.

6.3 Exploring the route template syntax

So far in this book we've looked at simple route templates consisting of fixed values, such as `/person` and `/test`, as well as using a basic route parameter such as `/fruit/{id}`. In this section we explore the full range of features available in route templates, such as default values, optional segments, and constraints.

6.3.1 Working with parameters and literal segments

Route templates have a rich, flexible syntax. Figure 6.5, however, shows a simple example, similar to ones you've already seen.

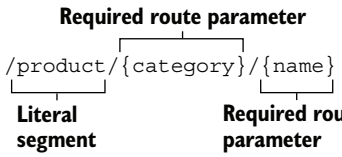

Figure 6.5 A simple route template showing a literal segment and two required route parameters

The routing middleware parses a route template by splitting it into *segments*. A segment is typically separated by the `/` character, but it can be any valid character.

DEFINITION Segments that use a character other than / are called *complex segments*. I generally recommend that you avoid them and stick to using / as a separator. Complex segments have some peculiarities that make them hard to use, so be sure to check the documentation at http://mng.bz/D4RE before you use them.

Each segment is either

- *A literal value* such as `product` in figure 6.5
- *A route parameter* such as `{category}` and `{name}` in figure 6.5

The request URL must match literal values exactly (ignoring case). If you need to match a particular URL exactly, you can use a template consisting only of literals.

TIP Literal segments in ASP.NET Core aren't case-sensitive.

Imagine that you have a minimal API in your application defined using

```
app.MapGet("/About/Contact", () => {/* */})
```

This route template, "`/About/Contact`", consists only of literal values, so it matches only the exact URL (ignoring case). None of the following URLs would match this route template:

- `/about`
- `/about-us/contact`
- `/about/contact/email`
- `/about/contact-us`

Route parameters are sections of a URL that may vary but are still a match for the template. You define them by giving them a name and placing them in braces, such as `{category}` or `{name}`. When used in this way, the parameters are required, so the request URL must have a segment that they correspond to, but the value can vary.

The ability to use route parameters gives you great flexibility. The simple route template `"/{category}/{name}"` could be used to match all the product-page URLs in an e-commerce application:

- `/bags/rucksack-a`—Where `category=bags` and `name=rucksack-a`
- `/shoes/black-size9`—Where `category=shoes` and `name=black-size9`

But note that this template would *not* map the following URLs:

- `/socks/`—No `name` parameter specified
- `/trousers/mens/formal`—Extra URL segment, `formal`, not found in route template

When a route template defines a route parameter and the route matches a URL, the value associated with the parameter is captured and stored in a dictionary of values associated with the request. These *route values* typically drive other behavior in the

endpoint and can be injected into the handlers (as you saw briefly in chapter 5) in a process called *model binding*.

> **DEFINITION** *Route values* are the values extracted from a URL based on a given route template. Each route parameter in a template has an associated route value, and the values are stored as a string pair in a dictionary. They can be used during model binding, as you'll see in chapter 7.

Literal segments and route parameters are the two cornerstones of ASP.NET Core route templates. With these two concepts, it's possible to build all manner of URLs for your application. In the remainder of section 6.3 we'll look at additional features that let you have optional URL segments, provide default values when a segment isn't specified, and place additional constraints on the values that are valid for a given route parameter.

6.3.2 Using optional and default values

In section 6.3.1 you saw a simple route template with a literal segment and two required routing parameters. Figure 6.6 shows a more complex route that uses several additional features.

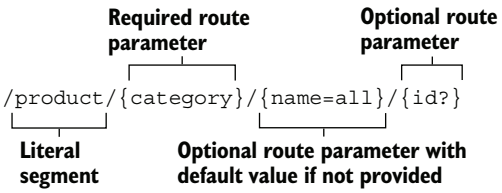

Figure 6.6 A more complex route template showing literal segments, named route parameters, optional parameters, and default values.

The literal `product` segment and the required `{category}` parameter are the same as those in in figure 6.6. The `{name}` parameter looks similar, but it has a default value specified for it by `=all`. If the URL doesn't contain a segment corresponding to the `{name}` parameter, the router will use the `all` value instead.

The final segment of figure 6.6, `{id?}`, defines an optional route parameter called `id`. This segment of the URL is optional. If this segment is present, the router captures the value for the `{id}` parameter; if the segment isn't there, the router doesn't create a route value for `id`.

You can specify any number of route parameters in your templates, and these values will be available to you for model binding. The complex route template shown in figure 6.6 allows you to match a greater variety of URLs by making `{name}` and `{id}` optional and by providing a default for `{name}`. Table 6.1 shows some of the URLs that this template would match and the corresponding route values that the router would set.

Note that there's no way to specify a value for the optional `{id}` parameter without also specifying the `{category}` and `{name}` parameters. You can put an optional parameter (that doesn't have a default) only at the end of a route template.

Table 6.1 URLs that would match the template of figure 6.7 and their corresponding route values

URL	Route values
/product/shoes/formal/3	category=shoes, name=formal, id=3
/product/shoes/formal	category=shoes, name=formal
/product/shoes	category=shoes, name=all
/product/bags/satchels	category=bags, name=satchels
/product/phones	category=phones, name=all
/product/computers/laptops/ABC-123	category=computers, name=laptops, id=ABC-123

Using default values allows you to have multiple ways to call the same URL, which may be desirable in some cases. Given the route template in figure 6.6, the following two URLs are equivalent:

- /product/shoes
- /product/shoes/all

Both URLs will execute the same endpoint handler, with the same route values of category=shoes and name=all. Using default values allows you to use shorter, more memorable URLs in your application for common URLs but still gives you the flexibility to match a variety of routes in a single template.

6.3.3 Adding additional constraints to route parameters

By defining whether a route parameter is required or optional and whether it has a default value, you can match a broad range of URLs with terse template syntax. Unfortunately, in some cases this approach ends up being a little too broad. Routing only matches URL segments to route parameters; it doesn't know anything about the data you're expecting those route parameters to contain. If you consider a template similar to the one in figure 6.6, "/{category}/{name=all}/{id?}", all of the following URLs would match:

- /shoes/sneakers/test
- /shoes/sneakers/123
- /Account/ChangePassword
- /ShoppingCart/Checkout/Start
- /1/2/3

These URLs are perfectly valid given the template's syntax, but some might cause problems for your application. These URLs have two or three segments, so the router happily assigns route values and matches the template when you might not want it to! These are the route values assigned:

- `/shoes/sneakers/test` has route values `category=shoes`, `name=sneakers`, and `id=test`.
- `/shoes/sneakers/123` has route values `category=shoes`, `name=sneakers`, and `id=123`.
- `/Account/ChangePassword` has route values `category=Account`, and `name=ChangePassword`.
- `/Cart/Checkout/Start` has route values `category=Cart`, `name=Checkout`, and `id=Start`.
- `/1/2/3` has route values `category=1`, `name=2`, and `id=3`.

Typically, the router passes route values to handlers through model binding, which you saw briefly in chapter 5 (and which chapter 7 discusses in detail). A minimal API endpoint defined as

```
app.MapGet("/fruit/{id}", (int id) => "Hello world!");
```

would obtain the `id` argument from the `id` route value. If the `id` route parameter ends up assigned a *noninteger* value from the URL, you'll get an exception when it's bound to the *integer* `id` parameter.

To avoid this problem, it's possible to add more *constraints* to a route template that must be satisfied for a URL to be considered a match. You can define constraints in a route template for a given route parameter by using : (colon). `{id:int}`, for example, would add the `IntRouteConstraint` to the `id` parameter. For a given URL to be considered a match, the value assigned to the `id` route value must be convertible to an integer.

You can apply a large number of route constraints to route templates to ensure that route values are convertible to appropriate types. You can also check more advanced constraints, such as that an integer value has a particular minimum value, that a string value has a maximum length, or that a value matches a given regular expression. Table 6.2 describes some of the available constraints. You can find a more complete list online in Microsoft's documentation at http://mng.bz/BmRJ.

Table 6.2 A few route constraints and their behavior when applied

Constraint	Example	Description	Match examples
`int`	`{qty:int}`	Matches any `integer`	`123, -123, 0`
`Guid`	`{id:guid}`	Matches any `Guid`	`d071b70c-a812-4b54-87d2-7769528e2814`
`decimal`	`{cost:decimal}`	Matches any `decimal` value	`29.99, 52, -1.01`
`min(value)`	`{age:min(18)}`	Matches `integer` values of 18 or greater	`18, 20`
`length(value)`	`{name:length(6)}`	Matches `string` values with a length of `6`	`Andrew,123456`

Table 6.2 A few route constraints and their behavior when applied *(continued)*

Constraint	Example	Description	Match examples
`optional int`	`{qty:int?}`	Optionally matches any `integer`	`123, -123, 0, null`
`optional int max(value)`	`{qty:int:max(10)?}`	Optionally matches any `integer` of 10 or less	`3, -123, 0, null`

TIP As you can see from table 6.2, you can also combine multiple constraints by separating the constraints with colons.

Using constraints allows you to narrow down the URLs that a given route template will match. When the routing middleware matches a URL to a route template, it interrogates the constraints to check that they're all valid. If they aren't valid, the route template isn't considered a match, and the endpoint won't be executed.

WARNING Don't use route constraints to validate general input, such as to check that an email address is valid. Doing so will result in `404` "Page not found" errors, which will be confusing for the user. You should also be aware that all these built-in constraints assume invariant culture, which may prove to be problematic if your application uses URLs localized for other languages.

Constraints are best used sparingly, but they can be useful when you have strict requirements on the URLs used by the application, as they can allow you to work around some otherwise-tricky combinations. You can even create custom constraints, as described in the documentation at http://mng.bz/d14Q.

Constraints and overlapping routes

If you have a well-designed set of URLs for your application, you'll probably find that you don't need to use route constraints. Route constraints are most useful when you have overlapping route templates.

Suppose that you have an endpoint with the route template `"/{number}/{name}"` and another with the template `"/{product}/{id}"`. When a request with the URL `/shoes/123` arrives, which template is chosen? Both match, so the routing middleware panics and throws an exception—not ideal.

Using constraints can fix this problem. If you update the first template to `"/{number:int}/{name}"`, the integer constraint means that the URL is no longer a match, and the routing middleware can choose correctly. Note, however, that the URL `/123/shoes` still matches both route templates, so you're not out of the woods.

Generally, you should avoid overlapping route templates like these, as they're often confusing and more trouble than they're worth. If your route templates are well defined so that each URL maps to a single template, ASP.NET Core routing will work without any difficulties. Sticking to the built-in conventions as far as possible is the best way to stay on the happy path!

We're coming to the end of our look at route templates, but before we move on, there's one more type of parameter to think about: the catch-all parameter.

6.3.4 Matching arbitrary URLs with the catch-all parameter

You've seen how route templates take URL segments and attempt to match them to parameters or literal strings. These segments normally split around the slash character, `/`, so the route parameters themselves won't contain a slash. What do you do if you need them to contain a slash or don't know how many segments you're going to have?

Imagine that you're building a currency-converter application that shows the exchange rate from one currency to one or more other currencies. You're told that the URLs for this page should contain all the currencies as separate segments. Here are some examples:

- `/USD/convert/GBP`—Show USD with exchange rate to GBP.
- `/USD/convert/GBP/EUR`—Show USD with exchange rates to GBP and EUR.
- `/USD/convert/GBP/EUR/CAD`—Show USD with exchange rates for GBP, EUR, and CAD.

If you want to support showing any number of currencies, as these URLs do, you need a way to capture everything after the `convert` segment. You could achieve this goal by using a catch-all parameter in the route template, as shown in figure 6.7.

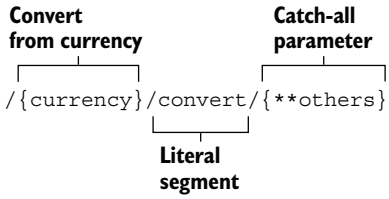

Figure 6.7 You can use catch-all parameters to match the remainder of a URL. Catch-all parameters may include the / character or may be an empty string.

You can declare catch-all parameters by using either one or two asterisks inside the parameter definition, as in `{*others}` and `{**others}`. These parameters match the remaining unmatched portion of a URL, including any slashes or other characters that aren't part of earlier parameters. They can also match an empty string. For the `USD/convert/GBP/EUR` URL, the value of the route value `others` would be the single string `"GBP/EUR"`.

> **TIP** Catch-all parameters are greedy and will capture the whole unmatched portion of a URL. Where possible, to avoid confusion, avoid defining route templates with catch-all parameters that overlap other route templates.

The one- and two-asterisk versions of the catch-all parameter behave identically when routing an incoming request to an endpoint. The difference occurs only when you're *generating* URLs (which we'll cover in the next section): the one-asterisk version URL encodes forward slashes, and the two-asterisk version doesn't. Typically, the round-trip behavior of the two-asterisk version is what you want.

> **NOTE** For examples and a comparison between the one and two-asterisk catch-all versions, see the documentation at http://mng.bz/rWyX.

You read that last paragraph correctly: mapping URLs to endpoints is only half of the responsibilities of the routing system in ASP.NET Core. It's also used to generate URLs so that you can reference your endpoints easily from other parts of your application.

6.4 Generating URLs from route parameters

In this section we'll look at the other half of routing: generating URLs. You'll learn how to generate a URL as a string you can use in your code and how to send redirect URLs automatically as a response from your endpoints.

One of the benefits and byproducts of using the routing infrastructure in ASP.NET Core is that your URLs can be somewhat fluid. You can change route templates however you like in your application—by renaming `/cart` to `/basket`, for example—and won't get any compilation errors.

Endpoints aren't isolated, of course; inevitably, you'll want to include a link to one endpoint in another. Trying to manage these links within your app manually would be a recipe for heartache, broken links, and `404` errors. If your URLs were hardcoded, you'd have to remember to do a find-and-replace operation with every rename!

Luckily, you can use the routing infrastructure to generate appropriate URLs dynamically at runtime instead, freeing you from the burden. Conceptually, this process is almost the exact reverse of the process of mapping a URL to an endpoint, as shown in figure 6.8. In the routing case, the routing middleware takes a URL, matches it to a route template, and splits it into route values. In the URL generation case, the generator takes in the route values and combines them with a route template to build a URL.

You can use the `LinkGenerator` class to generate URLs for your minimal APIs. You can use it in any part of your application, so you can use it in middleware and any other services too. `LinkGenerator` has various methods for generating URLs, such as `GetPathByPage` and `GetPathByAction`, which are used specifically for routing to Razor Pages and MVC actions, so we'll look at those in chapter 14. We're interested in the methods related to named routes.

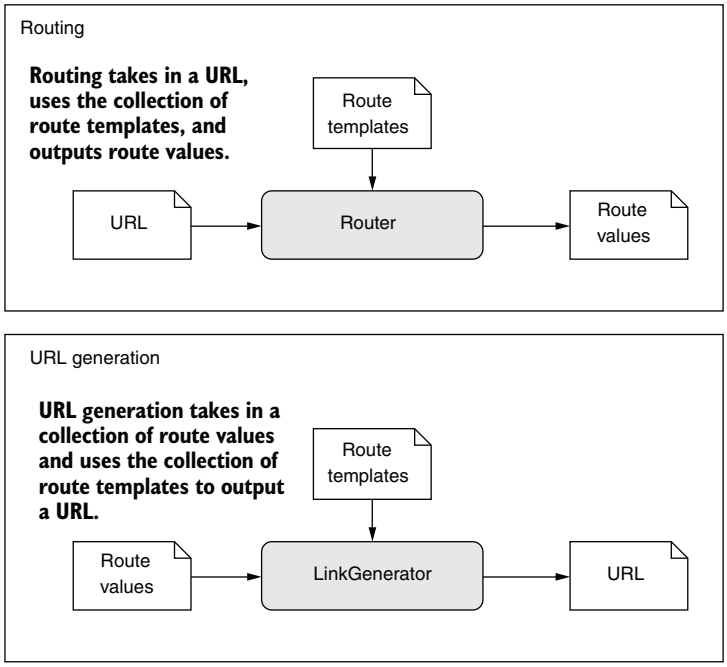

Figure 6.8 A comparison between routing and URL generation. Routing takes in a URL and generates route values, but URL generation uses route values to generate a URL.

6.4.1 Generating URLs for a minimal API endpoint with LinkGenerator

You'll need to generate URLs in various places in your application, and one common location is your minimal API endpoints. The following listing shows how you could generate a link to one endpoint from another by annotating the target endpoint with a name and using the `LinkGenerator` class.

Listing 6.2 Generating a URL `LinkGenerator` and a named endpoint

```
app.MapGet("/product/{name}", (string name) => $"The product is {name}")
    .WithName("product");

app.MapGet("/links", (LinkGenerator links) =>
{
    string link = links.GetPathByName("product",
        new { name = "big-widget"});
    return $"View the product at {link}";
});
```

The endpoint echoes the name it receives in the route template.

Gives the endpoint a name by adding metadata to it

References the LinkGenerator class in the endpoint handler

Creates a link using the route name "product" and provides a value for the route parameter

Returns the value "View the product at /product/big-widget"

The `WithName()` method adds metadata to your endpoints so that they can be referenced by other parts of your application. In this case, we're adding a name to the endpoint so we can refer to it later. You'll learn more about metadata in chapter 11.

> **NOTE** Endpoint names are case-sensitive (unlike the route templates themselves) and must be globally unique. Duplicate names cause exceptions at runtime.

The `LinkGenerator` is a service available anywhere in ASP.NET Core. You can access it from your endpoints by including it as a parameter in the handler.

> **NOTE** You can reference the `LinkGenerator` in your handler because it's registered with the dependency injection container automatically. You'll learn about dependency injection in chapters 8 and 9.

The `GetPathByName()` method takes the name of a route and, optionally, route data. The route data is packaged as key-value pairs into a single C# anonymous object. If you need to pass more than one route value, you can add more properties to the anonymous object. Then the helper will generate a path based on the referenced endpoint's route template.

Listing 6.2 shows how to generate a path. But you can also generate a complete URL by using the `GetUriByName()` method and providing values for the host and scheme, as in this example:

```
links.GetUriByName("product", new { Name = "super-fancy-widget"},
    "https", new HostString("localhost"));
```

Also, some methods available on `LinkGenerator` take an `HttpContext`. These methods are often easier to use in an endpoint handler, as they extract ambient values such as the scheme and hostname from the incoming request and reuse them for URL generation.

> **WARNING** Be careful when using the `GetUriByName` method. It's possible to expose vulnerabilities in your app if you use unvalidated host values. For more information on host filtering and why it's important, see this post: http://mng.bz/V1d5.

In listing 6.2, as well as providing the route name, I passed in an anonymous object to `GetPathByName`:

```
string link = links.GetPathByName("product", new { name = "big-widget"});
```

This object provides additional route values when generating the URL, in this case setting the `name` parameter to `"big-widget"`.

If a selected route explicitly includes the defined route value in its definition, such as in the `"/product/{name}"` route template, the route value will be used in the URL path, resulting in `/product/big-widget`. If a route doesn't contain the route value

explicitly, as in the `"/product"` template, the route value is appended to the query string as additional data. as in `/product?name=big-widget`.

6.4.2 Generating URLs with IResults

Generating URLs that link to other endpoints is common when you're creating a REST API, for example. But you don't always need to display URLs. Sometimes, you want to redirect a user to a URL automatically. In that situation you can use `Results.RedirectToRoute()` to handle the URL generation instead.

> **NOTE** Redirects are more common with server-rendered applications such as Razor Pages, but they're perfectly valid for API applications too.

Listing 6.3 shows how you can return a response from an endpoint that automatically redirects a user to a different named endpoint. The `RedirectToRoute()` method takes the name of the endpoint and any required route parameters, and generates a URL in a similar way to `LinkGenerator`. The minimal API framework automatically sends the generated URL as the response, so you never see the URL in your code. Then the user's browser reads the URL from the response and automatically redirects to the new page.

Listing 6.3 Generating a redirect URL using `Results.RedirectToRoute()`

```
app.MapGet("/test", () => "Hello world!")
    .WithName("hello");

app.MapGet("/redirect-me",
    () => Results.RedirectToRoute("hello"))
```

Annotates the route with the name "hello"

Generates a response that sends a redirect to the "hello" endpoint

By default, `RedirectToRoute()` generates a `302 Found` response and includes the generated URL in the Location response header. You can control the status code used by setting the optional parameters `preserveMethod` and `permanent` as follows:

- `permanent=false, preserveMethod=false`—`302 Found`
- `permanent=true, preserveMethod=false`—`301 Moved Permanently`
- `permanent=false, preserveMethod=true`—`307 Temporary Redirect`
- `permanent=true, preserveMethod=true`—`308 Permanent Redirect`

> **NOTE** Each of the redirect status codes has a slightly different semantic meaning, though in practice, many sites simply use `302`. Be careful with the permanent move status codes; they'll cause browsers to never call the original URL, always favoring the redirect location. For a good explanation of these codes (and the useful `303 See Other` status code), see the Mozilla documentation at http://mng.bz/x4GB.

As well as redirecting to a specific endpoint, you can redirect to an arbitrary URL by using the `Results.Redirect()` method. This method works in the same way as `RedirectToRoute()` but takes a URL instead of a route name and can be useful for redirecting to external URLs.

Whether you're generating URLs by using `LinkGenerator` or `RedirectToRoute()`, you need to be careful in these route generation methods. Make sure to provide the correct endpoint name and any necessary route parameters. If you get something wrong—if you have a typo in your endpoint name or forget to include a required route parameter, for example—the URL generated will be `null`. Sometimes it's worth checking the generated URL for `null` explicitly to make sure that there are no problems.

6.4.3 Controlling your generated URLs with RouteOptions

Your endpoint routes are the public surface of your APIs, so you may well have opinions on how they should look. By default, `LinkGenerator` does its best to generate routes the same way you define them; if you define an endpoint with the route template `/MyRoute`, `LinkGenerator` generates the path `/MyRoute`. But what if that path isn't what you want? What if you'd rather have `LinkGenerator` produce prettier paths, such as `/myroute` or `/myroute/`? In this section you'll learn how to configure URL generation both globally and on a case-by-case basis.

> **NOTE** Whether to add a trailing slash to your URLs is largely a question of taste, but the choice has some implications in terms of both usability and search results. I typically choose to add trailing slashes for Razor Pages applications but not for APIs. For details, see http://mng.bz/Ao1W.

When ASP.NET Core matches an incoming URL against your route templates by using routing, it uses a case-insensitive comparison, as you saw in chapter 5. So if you have a route template `/MyRoute`, requests to `/myroute`, `/MYROUTE`, and even `/myROUTE` match. But when generating URLs, `LinkGenerator` needs to choose a single version to use. By default, it uses the same casing that you defined in your route templates. So if you write

```
app.MapGet("/MyRoute", () => "Hello world!").WithName("route1");
```

`LinkGenerator.GetPathByName("route1")` returns `/MyRoute`.

Although that's a good default, you'd probably prefer that all the links generated by your app be consistent. I like all my links to be lowercase, regardless of whether I accidentally failed to make my route template lowercase.

You can control the route generation rules by using `RouteOptions`. You configure the `RouteOptions` for your app using the `Configure<T>` extension method on `WebApplicationBuilder.Services`, which updates the `RouteOptions` instance for the app using the configuration system.

> **NOTE** You'll learn all about the configuration system and the `Configure<T>` method in chapter 10.

`RouteOptions` contains several configuration options, as shown in listing 6.4. These settings control whether the URLs your app generates are forced to be lowercase, whether the query string should also be lowercase, and whether a trailing slash (`/`)

should be appended to the final URLs. In the listing, I set the URL to be lowercased, for the trailing slash to be added, and for the query string to remain unchanged.

> **NOTE** In listing 6.4 the whole path is lowercased, including any route parameter segments such as `{name}`. Only the query string retains its original casing.

Listing 6.4 Configuring link generation using `RouteOptions`

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.Configure<RouteOptions>(o =>
{
    o.LowercaseUrls = true;
    o.AppendTrailingSlash = true;
    o.LowercaseQueryStrings = false;
});

WebApplication app = builder.Build();

app.MapGet("/HealthCheck", () => Results.Ok()).WithName("healthcheck");
app.MapGet("/{name}", (string name) => name).WithName("product");

app.MapGet("/", (LinkGenerator links) =>
new []
{
    links.GetPathByName("healthcheck"),
    links.GetPathByName("product",
        new { Name = "Big-Widget", Q = "Test"})
});

app.Run();
```

Configures the RouteOptions used for link generation

All the settings default to false.

Returns /healthcheck/

Returns /big-widget/?Q=Test

Whatever default options you choose, you should try to use them throughout your whole app, but in some cases that may not be possible. You might have a legacy API that you need to emulate, for example, and can't use lowercase URLs. In these cases, you can override the defaults by passing an optional `LinkOptions` parameter to `LinkGenerator` methods. The values you set in `LinkOptions` override the default values set in `RouteOptions`. Generating a link for the app in listing 6.4 by using

```
links.GetPathByName("healthcheck",
    options: new LinkOptions
    {
        LowercaseUrls = false,
        AppendTrailingSlash = false,
    });
```

would return the value `/HealthCheck`. Without the `LinkOptions` parameter, `GetPathByName` would return `/healthcheck/`.

Congratulations—you've made it all the way through this detailed discussion of routing! Routing is one of those topics that people often get stuck on when they come to building an application, which can be frustrating. We'll revisit routing when we

look at Razor Pages in chapter 14 and web API controllers in chapter 20, but rest assured that this chapter has covered all the tricky details!

In chapter 7 we'll dive into model binding. You'll see how the route values generated during routing are bound to your endpoint handler parameters and, perhaps more important, how to validate the values you're provided.

Summary

- Routing is the process of mapping an incoming request URL to an endpoint that executes to generate a response. Routing provides flexibility to your API implementations, enabling you to map multiple URLs to a single endpoint, for example.
- ASP.NET Core uses two pieces of middleware for routing. The `EndpointRoutingMiddleware` and the `EndpointMiddleware`. `WebApplication` adds both pieces of middleware to your pipeline by default, so typically, you don't add them to your application manually.
- The `EndpointRoutingMiddleware` selects which endpoint should be executed by using routing to match the request URL. The `EndpointMiddleware` executes the endpoint. Having two separate middleware components means that middleware placed between them can react based on the endpoint that will execute when it reaches the end of the pipeline.
- Route templates define the structure of known URLs in your application. They're strings with placeholders for variables that can contain optional values and map to endpoint handlers. You should think about your routes carefully, as they're the public surface of your application.
- Route parameters are variable values extracted from a request's URL. You can use route parameters to map multiple URLs to the same endpoint and to extract the variable value from the URL automatically.
- Route parameters can be optional and can use default values when a value is missing. You should use optional and default parameters sparingly, as they can make your APIs harder to understand, but they can be useful in some cases. Optional parameters must be the last segment of a route.
- Route parameters can have constraints that restrict the possible values allowed. If a route parameter doesn't match its constraints, the route isn't considered to be a match. This approach can help you disambiguate between two similar routes, but you shouldn't use constraints for validation.
- Use a catch-all parameter to capture the remainder of a URL into a route value. Unlike standard route parameters, catch-all parameters can include slashes (`/`) in the captured values.
- You can use the routing infrastructure to generate URLs for your application. This approach ensures that all your links remain correct if you change your endpoint's route templates.

- The `LinkGenerator` can be used to generate URLs from minimal API endpoints. Provide the name of the endpoint to link to and any required route values to generate an appropriate URL.
- You can use the `RedirectToRoute` method to generate URLs while also generating a redirect response. This approach is useful when you don't need to reference the URL in code.
- By default, URLs are generated using the same casing as the route template and any supplied route parameters. Instead, you can force lowercase URLs, lowercase query strings, and trailing slashes by customizing `RouteOptions`, calling `builder.Services.Configure<RouteOptions>()`.
- You can change the settings for a single URL generation by passing a `LinkOptions` object to the `LinkGenerator` methods. These methods can be useful when you need to differ from the defaults for a single endpoint, such as when you're trying to match an existing legacy route.